Console I/O, Compilation and Parsing

CSC 230 : C and Software Tools

NC State Department of Computer Science

Topics for Today

- Console I/O
  - Character I/O
  - printf()
- The C Standard and Coding Style
- Program Execution in Java and C
- Using gcc
- Meet the Preprocessor
- Tokenization

Console I/O In C

- In C, I/O is provided by functions in the *standard library*
  - This library is expected on all platforms
- To use the I/O parts of the standard library, you need to include the header file:

```
#include <stdio.h>
```

- We'll also get some use out of:

```
#include <stdlib.h>
```

- These are *preprocessor directives*, telling the preprocessor to get these files and compile them along with our source code.

Streams

- A *stream* is a file or device we can read or write
- Just like in Java, a C program starts with three streams it can use
  - *Standard input* (input typed at the terminal)
  - *Standard output* (output to the terminal)
  - *Standard error* (more output to the terminal)
- Reading and writing to the terminal looks just like reading or writing a file
  - We can even signal the End-Of-File condition on standard input:
    - Type CTRL-D in Linux
    - CTRL-Z in windows.

Redirecting Standard Streams

- We can redirect these streams to or from actual files (without the program even noticing)
  - We won't learn about file I/O for a while, but this will let us get by without it.
- From the terminal, you can redirect standard input from a file

```
$ myProgram < input_file.txt
```

- … or standard output to a file.

```
$ myProgram > output_file.txt
```

Reading just one Character

- stdio.h provides a function
  `int getchar()`

It returns an int, the next character read.

It doesn't take any parameters.

- Returns the next character read,
- … or the constant EOF if there's no more input.
- That's why it's return type is int instead of char

Writing just one Character

- stdio.h also provides

  `int putchar( int c )`

Returns the character you just wrote, or EOF if it can't.

The character you want to write.

Formatted Output

- The printf() function is good for generating formatted output
- You probably saw a similar function in Java. It works like:

```
printf( “value: %6.2f\n”, 3.1415926 );
```

This is a format string. Most of it gets printed literally.

But not parts like this. This is a *format specification*.

Each format specification says how to print one of the remaining parameters.

Making Sense of Format Specifications

This says "Here comes a format specification"

Format Conversion
This says what to print, here it's a double.

`%6.2f`

Minimum Field Width
This part is optional. It says "use at least 6 characters of output".

Precision
This part is optional.
It says "output two fractional digits".

Format Specification Examples

- There are lots of ways to print a value like 33.3:

Format Specification	Output
%7.1f	`   33.3`
%14.10f	` 33.3000000000`
%20.20f	`33.29999999999999715783`

3 spaces

1 space

no spaces

Format Specification Overview

- We can print more than just double values

Format Specifier	Output
%c	A single char
%d	A decimal integer
%x	A decimal integer in hexadecimal
%o	A decimal integer in octal
%f	A float or double
%e	A float or double in scientific notation
%s	A string
%zd	The size of some memory region (a value of type size_t)

Fun with getchar() and putchar()

- Reading and counting characters.
  - A good example.
  - A bad example.
- Let’s write a program, echoline, to read and echo a line of text
- Let’s use redirection to have it to read or write from a file

It’s About Standards

- C has developed since it was first created
  - K&R C (informal standard)
  - C89 (ISO standard)
  - C99 (ISO standard)
- We’ll target C99 in this course
  - But, in the real world, you sometimes have to read or write for an older standard
- What do we get with C99?

C89 vs C99

- We get to use // comments
- We get the _Bool type (and bool with stdbool.h)
  - And some other types (long long, complex)
- We get variable-length arrays
- We can declare variables where we need them (not just at the top of functions or blocks)
- Support for wide characters
- Several new library functions (for math, wide characters, etc.)
- Some compiler hints (inline, restrict)

Coding Style Conventions

- There are lots of ways you **can** write a working program
- But there's a difference between what you can do and what you should do
- Consider this submission from the first International Obfuscated C Code Contest:

```
int i;main(){for(;i["]<i;++i){--i;}"];read('-'-'-',i+++"hell\
o, world!\n",'/'/'/'));}read(j,i,p){write(j/p+p,i---j,i/i);}
```

  – I'm told it's a "Hello World" program

Coding Style Conventions

- Or this one from the 1993 contest:

```
O5(O2,O7,O3)char**O7;{return!(O2+=~01+01)?00:!(O2-=02>01)?printf("\045\157\012"
,O5(012,O7+01,00)):!(O2-=02>>01)?(**O7<=067&&**O7>057?O5(O3,O7,*(*O7)++-060+010
   *O3):O3                     ):!(O2      -=-O3-      ~O3)?       (072>**
   O7&&060                     <=**O7      ?O5(04      ,O7,012     *O3-060
    +*(*O7                    )++):O3      ):!(O2      -=!O3+      !!O3)?(
    **O7>057                  &&**O7       <=071?      O5(O5,      O7,*(*
     O7)+++                   O3*020       -060):      **O7<=      0106&&
     00101<=                 **O7?O5       (O5,O7      ,020*O3     +*(*O7)
      ++-067)               :0140<**       O7&&**      O7<0147     ?O5(O5,
      O7,-0127              +*(*O7         )+++020     *O3):O3     ):!(
       O2-=O2-             01)?(**         O7==050     ?050**      ++*O7,
       O5(013,             O7,O5(          012,O7      ,00)):*     *O7<056
        &&054<*           *O7?055          **++*       O7,-O5(     06,O7,
        00):054           >**O7&&          052<**      O7?050*     *(*O7)
         ++,O5(06        ,O7,00            ):!(**      O7^0170     )||!(
         0130^**         O7)?*++           *O7,O5      (O5,O7      ,00):*
          *O7==0144     ||**O7             ==0104      ?++*O7      ,O5(04,
           O7,00):      O5(O3              ,O7,00      )):!--      O2?(*
           *O7==052    ?O5(O7              ,O7,O3*     (*++*O7     ,O5(06
           ,O7,00)     )):!(               045-**      O7)?O5(     O7,O7,
            O3%(O3+( *O7)++,               O5(06,      O7,00)      )):!(**
             O7^057)?O5(O7,                O7,O3/(     O3-*++      *O7,O5(
             06,O7,00))):O3                ):!(O2      +=01-O2     )?O5(O7
             ,O7,O5(06,O7,                 00)):!(     O2+=-O2/    O2)?(!(*
*O7-053)?O5(011,O7,O3+(++*O7,O5(010,O7,00))):!(055^**O7)?O5(011,O7,O3-(O3+*(*O7
)++,O5(0010,O7,00))):O3):!(O2-=0563&0215)?O5(011,O7,O5(010,O7,00)):(++*O7,O3);}
```

Coding Style Conventions

- These examples are deliberately hard to read and understand
- Normally, this is the opposite of what we want
- We will adopt some coding style conventions, rules for:
  - Naming conventions
  - Spacing and indentation
  - Where important comments go and what they contain
- Fortunately, editors can often help us with this.

Comments in C

- C has block-style comments, like Java;

```
int w = 25;        /* Output width */
int h = 10;        /* Output height */
```

- Handy for large comments
- Or for commenting out blocks of code ... but be careful, you can't nest comments:

```
/* I don't need these variables now
  int w = 25;        /* Output width */
  int h = 10;        /* Output height */
*/
```

Comments in C

- C99 lets us use to-end-of-line comments.

```c
int w = 25;         // Output width
int h = 10;         // Output height
```

- You can even use javadoc-style comments
  - But, you'll need a tool (e.g., doxygen) to do something useful with them.

```c
/**
  * This is the best function ever.
  * @author bill
  */
int f( int x, int y ) {
```

CSC 230 Style Guidelines

- A Javadoc-style block comment at the top of each source file
  - With a @file tag giving the filenmane
  - And an @author tag with name and unity ID.
  - And a brief description of what the program does
- A Javadoc-style block comment at the top of each function:
  - A sentence or two describing the function's purpose
  - @param tags for each parameter
  - @return tag for the return value
  - @pre and @post for pre- and post-conditions not already described as inputs/outputs.
  - A @sideeffect tag for any other side effects.

Omit any of these if they're not needed.

CSC 230 Style Guidelines

- A good comment on each constant, global variable and type definition.
- Magic numbers, avoid bare constants for:
  - **Any value that could have an explanation**

```
area = radius * radius * 3.1415926;
```

  - **Any potentially tunable parameter**

```
score += 350;
```

  - **Any value that needs to occur at multiple points in the code**

```
for ( int i = 0; i <= 99; i++ )
  …;
```

```
for ( int j = 99; j >= 0; j++ )
  …;
```

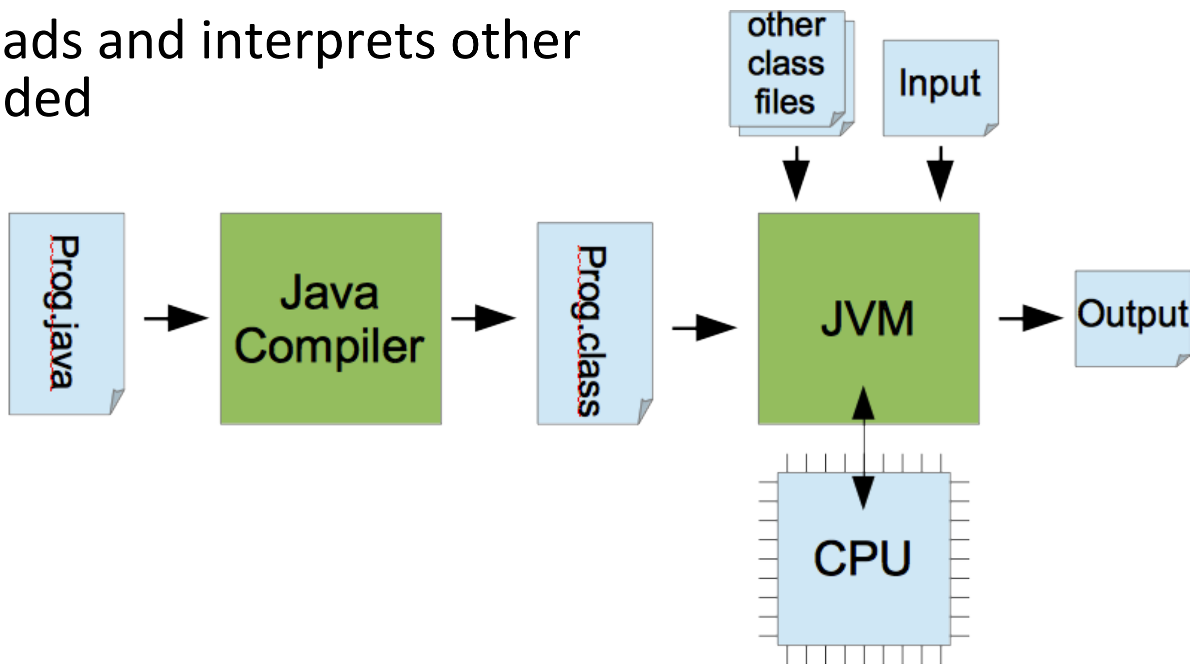
  - We'll learn how to define named constants soon

CSC 230 Style Guidelines

- Curly bracket placement
  - For function definitions, it goes on the next line (to make functions stand out)
  - For everything else, it goes on the same line
- Indentation
  - No hard tabs, just indent with spaces (why?)
  - Indent using any number of spaces you want, 2 spaces, 3 spaces … maybe 4 spaces.
  - But, be consistent.
- Just one statement per line

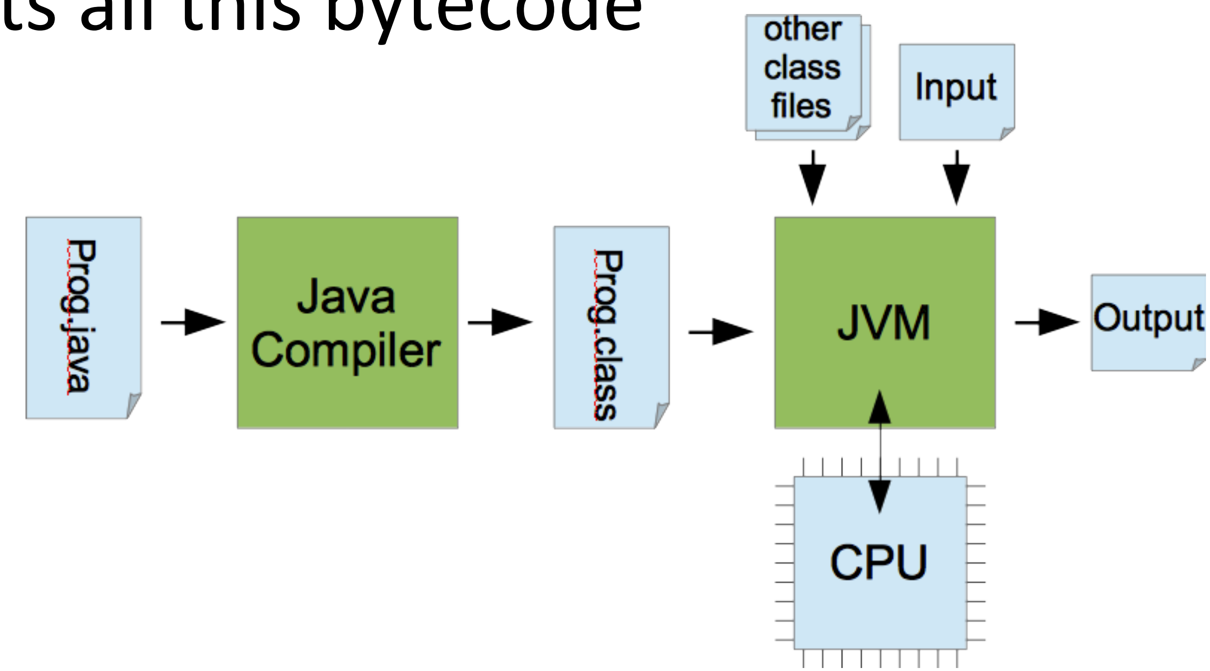
Executing Java Programs

- Java source code is *compiled* into Java class file containing *bytecode*
  - A platform-independent, intermediate representation
- To run it, we need an interpreter, the Java Virtual Machine
  - Takes a class file as input, runs native machine code to simulate each bytecode instruction
  - Automatically loads and interprets other class files as needed



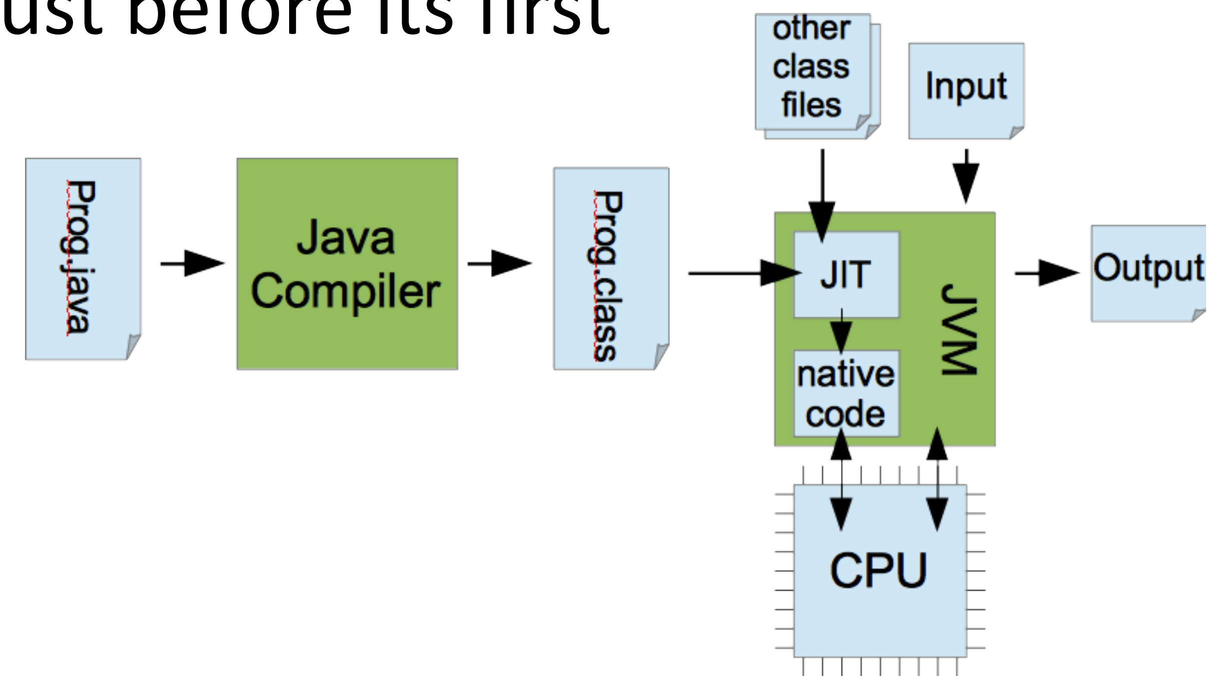
Executing Java Programs

- This is great. The class files for our compiled program are platform independent.
- But, some extra overhead is incurred as the JVM interprets all this bytecode



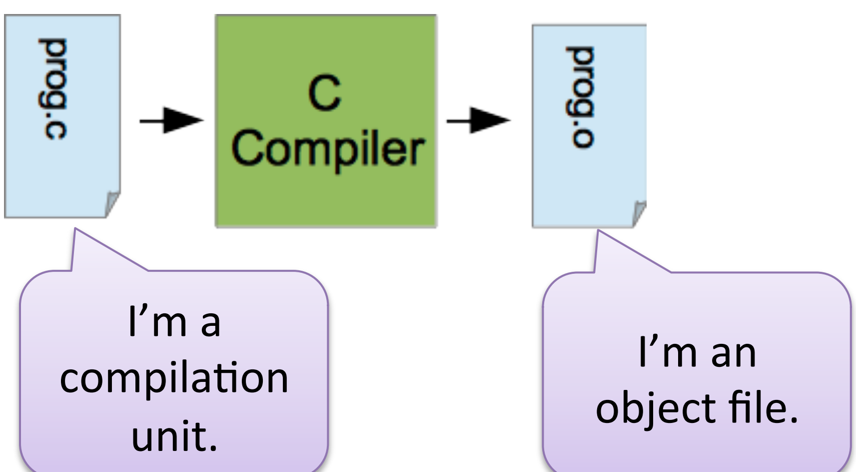
Executing Java Programs

- With Just-In-Time compilation, Java can get closer to native processor speeds
- Each method is compiled to native machine instructions just before its first execution



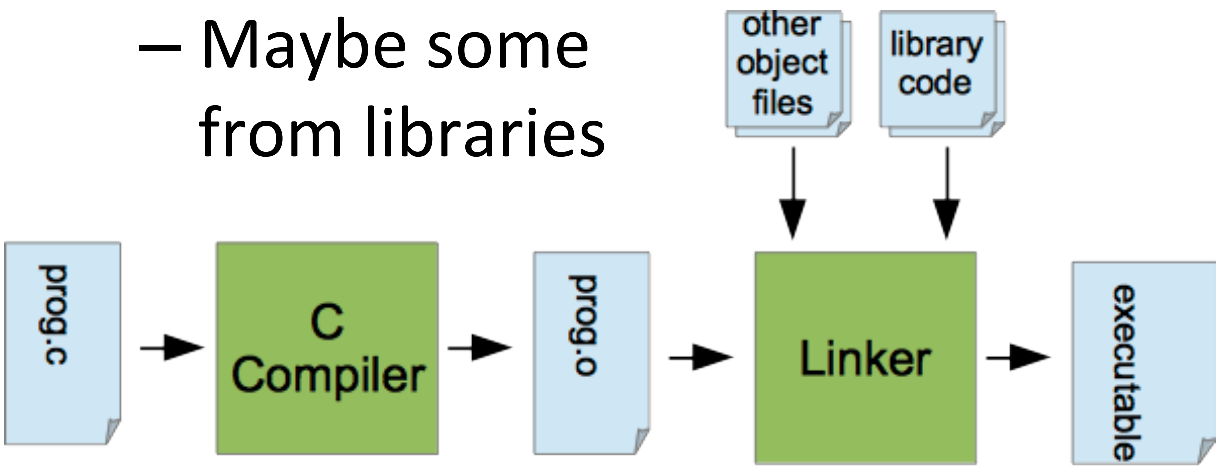
Executing C Programs

- A C Compiler generates native machine code for the target processor
- One *compilation unit* generates one *object file*



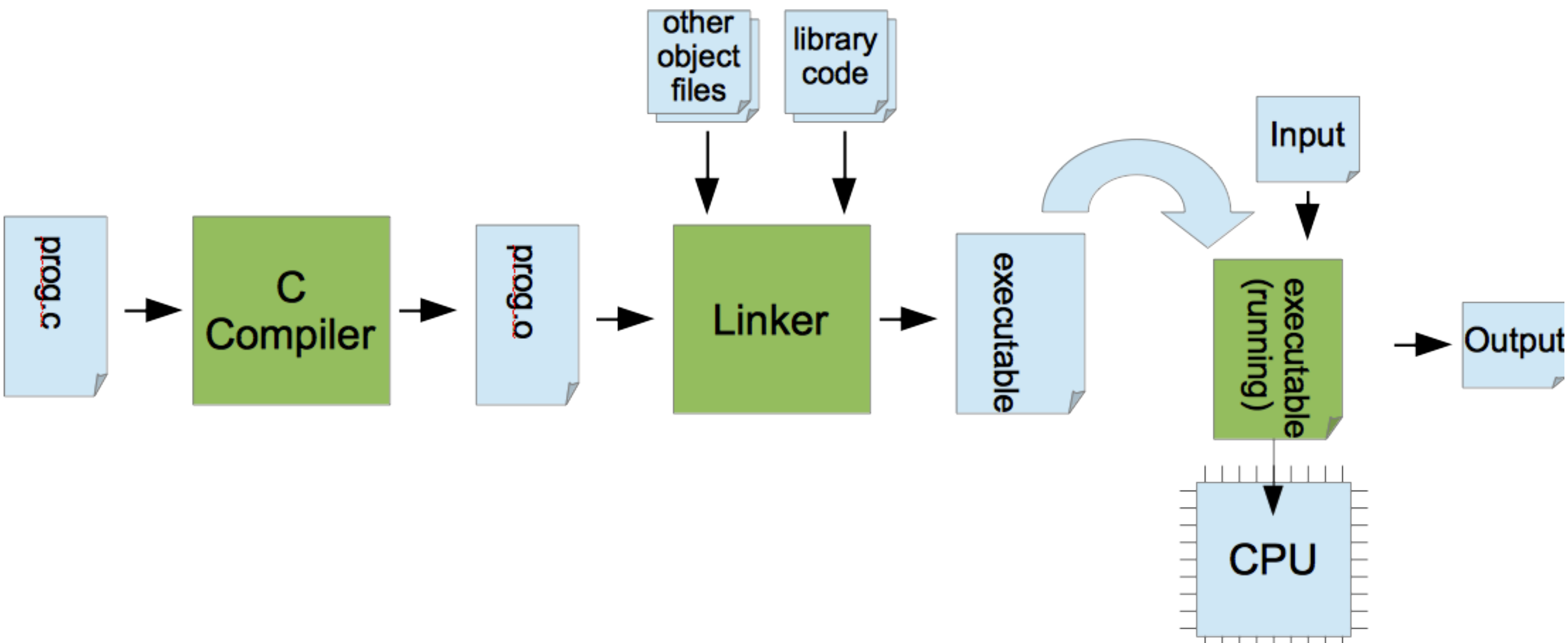
Executing C Programs

- A linker combines object files to create an executable program
  - Maybe some other objects we wrote
  - Maybe some from libraries



Executing C Programs

- The executable is ready to run
- Just load it into memory and start running at the top of main()

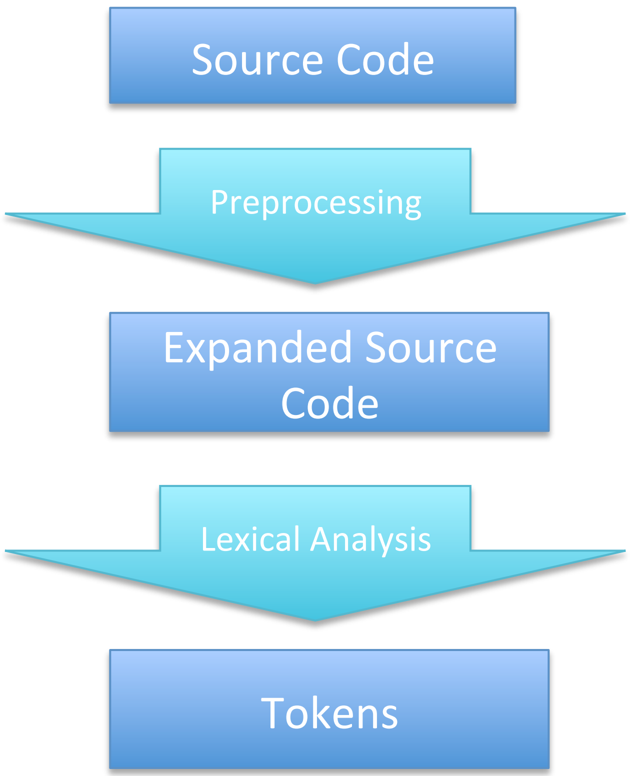


Compiled vs Interpreted

- Each approach has advantages.
- What do you think? Would compiled code (compiled to the native instruction set) execute faster than interpreted?
- Which would offer better support for debugging and error messages?
- Which would offer greater platform independence?
- Which would offer more opportunities for code analysis and optimization?

Steps in C Compilation

- Really, generating an executable has more steps than you might expect.

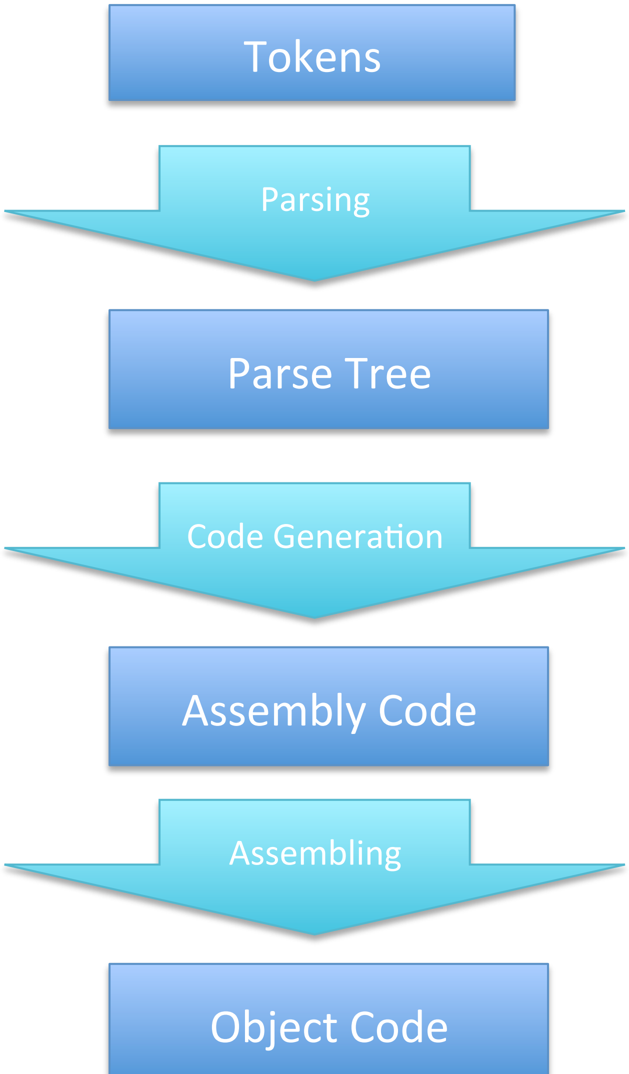



```
#define N 3
length += base * N;
```

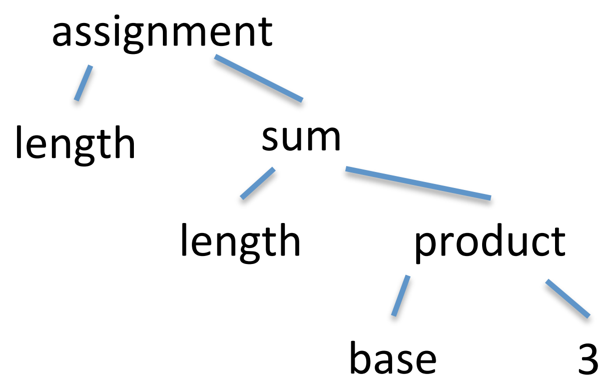
```
length += base * 3;
```

```
length   +=   base    *   3   ;
```

Steps in C Compilation



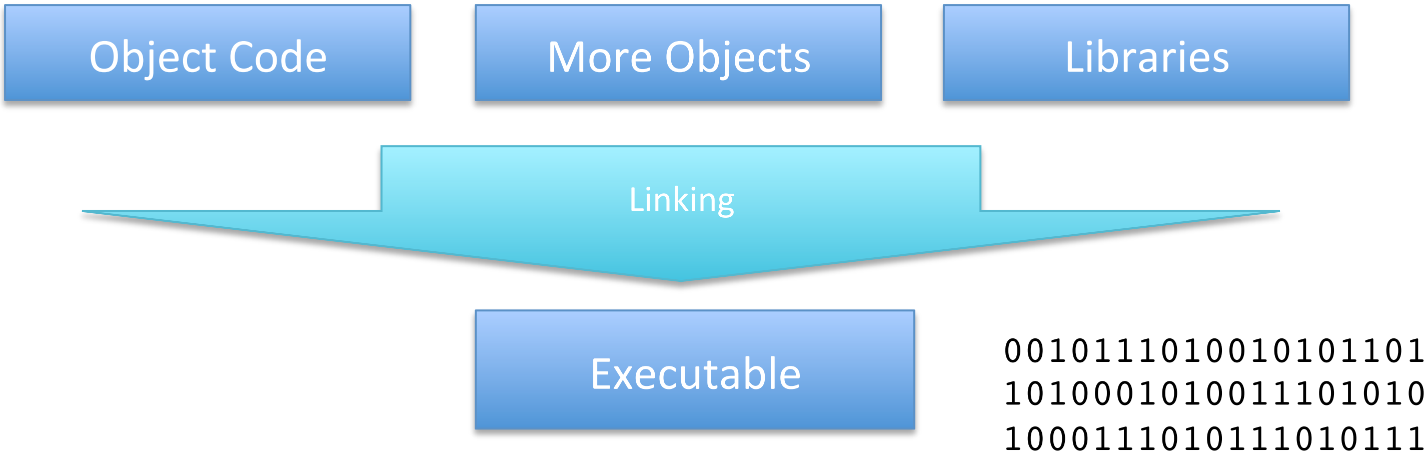
length += base * 3 ;




```
mov ebx, base
imul ebx, 3
iadd ebx, length
mov length, ebx
```

0010111010010101101

Steps in C Compilation



Using the gcc Compiler

- You already have a template for compiling with gcc

```
$ gcc -Wall -std=c99 X.c -o X
```

- We already talked about the -Wall, -std=c99 and -o options
- gcc will complain less if you omit the -Wall option
  - But, why would you want to do this? You want the compiler to do as much for you as it can.
- gcc has lots of other options

More Useful gcc Options

`-c`	Just compile the source file, don't link.
`-E`	Just run the preprocessor, don't compile.
`--version`	Report the version number
`-g`	Include additional information to help with debugging.
`-O, -O1, –O2 …`	Enable various levels of optimization
`-D` *`name`*	Define *name* as a preprocessor macro, with a value of 1 (used for conditional compilation)
`-l`*`lib`*	During linking, link with the library named *lib*.
`-lm`	Link with the math library (for functions like sqrt())

The Preprocessor

- The preprocessor operates on the source code before the compiler even sees it.
- Performs basic text operations
  - *Includes headers* : inserting code that enables use of code from other components
  - *Expands macros* : replacing macro names with corresponding definitions
  - More things we'll learn about later
- Lines starting with # are *preprocessor directives*
  - instructions processed (and removed) by the preprocessor

Preprocessor Constants

- Preprocessor macros give us a way to define named constants:

```
#define SIZE 25
```

Replace occurrences of this …

… with this.

Be careful, you probably don't want a semi-colon here.

```
for ( int i = 0; i < SIZE; i++ )
  …
```

```
for ( int i = 0; i < 25; i++ )
  …
```

Looking for Tokens

- The compiler has to break the source into *tokens*
- This is called *lexical analysis* or *scanning*
- A token can be:
  - An *identifier* (e.g., a variable or a function name)
  - A *keyword* (e.g., void or while)
  - A *literal value* (e.g., 3.1415, or “Hello World”)
  - An *operator* (e.g., *, ++ or >=)
  - An *explicit separator* (e.g., (, } or ;)
- White space between the tokens is ignored
  (except, of course, that it can separate tokens)

Fun with Lexical Analysis

- What are the tokens in:
  `a + ++b >= c-- -d`
- How about now:
  `a+++b>=c---d`
- There are lots of ways this could be parsed:

```
a ++ + b >= c - -- d   ?

a ++ +b > = c- - -d   ?

a + + + b > = c - - - d   ?
```

- This isn't about precedence.
  - We can't even think about precedence until we know what the operators are.

The Scanner is Greedy

- The scanner works from left to right, grabbing the longest token it can
  - This is called *maximal munch*
- So, for our example:

`a +++b>=c---d`	(because a+ isn't a token)
`a ++ +b>=c---d`	(because +++ isn't a token)
`a ++ + b>=c---d`	(because +b isn't a token)
`a ++ + b >=c---d`	(because b> isn't a token)
`a ++ + b >= c---d`	(because >=c isn't a token)
`a ++ + b >= c ---d`	(because c- isn't a token)
`a ++ + b >= c -- -d`	(because --- isn't a token)
`a ++ + b >= c -- - d`	(because -d isn't a token)

Be the Scanner

- In the following expression, how many tokens are there?

```
----xy+-=x++y*/z;
```

- What are they?
- This expression wouldn't parse, but we can still talk about what the scanner would do with it.